NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PHYSICAL BASED TOOLKIT FOR REAL-TIME DISTRIBUTED VIRTUAL WORLD

by

Henry Tong Ong

September 1995

Thesis Advisor: David R. Pratt
Thesis Co-Advisor: John S. Falby

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Master's Thesis

4. TITLE AND SUBTITLE
PHYSICAL BASED TOOLKIT FOR REAL-TIME DISTRIBUTED VIRTUAL WORLD

5. FUNDING NUMBERS

6. AUTHOR(S)
Ong, Henry T.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis addresses three deficiencies in the NPSNET simulated world. First, although a full set of algorithms have been defined for Dead Reckoning (DR) entities in a distributed simulation, NPSNET only implements a few simple linear algorithms. Second, NPSNET lacks a set of physically-based models for munition trajectories having, currently, only algorithms for the bullet and bomb. Third, NPSNET lacks physically-based models for engine power curves using, instead, a simple linear approximation.

The purpose of this thesis work is to implement an object-oriented programming toolkit which corrects these deficiencies. The code, in C++, utilizes class hierarchies. The toolkit implements the nine class hierarchies of DR algorithms described by the Advanced Research Projects Agency for the Distributed Interactive Simulation standard. The toolkit also provides treatment of a physically-based class hierarchy for munitions trajectories. In addition, a physically-based, mathematical model for the engines class was implemented.

As a result, a set of DR algorithms have been built to predict the position of simulated entities in all cases. The munitions class implements trajectories for a variety of projectiles. With this arsenal, future versions of NPSNET will be more realistic. The engine class, with new mathematical models, far more realistically represents engine behaviors than the current linear approximation. In summation, the implementation of this toolkit dovetails very well with the needs of NPSNET, and will be integrated into future releases.

14. SUBJECT TERMS	15. NUMBER OF PAGES
DIS, Simulation, Virtual World, Virtual Environment, Munition, Engine	54
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

PHYSICAL BASED TOOLKIT FOR REAL-TIME DISTRIBUTED VIRTUAL WORLD

Henry Tong Ong
Lieutenant, United States Navy
B.S., University of Washington, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1995

Author:

_____

Henry Tong Ong

Approved by:

_____

David R. Pratt, Thesis Advisor

_____

John S. Falby, Thesis Co-Advisor

_____

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

This thesis addresses three deficiencies in the NPSNET simulated world. First, although a full set of algorithms have been defined for Dead Reckoning (DR) entities in a distributed simulation, NPSNET only implements a few simple linear algorithms. Second, NPSNET lacks a set of physically-based models for munition trajectories having, currently, only algorithms for the bullet and bomb. Third, NPSNET lacks physically-based models for engine power curves using, instead, a simple linear approximation.

The purpose of this thesis work is to implement an object-oriented programming toolkit which corrects these deficiencies. The code, in C++, utilizes class hierarchies. The toolkit implements the nine class hierarchies of DR algorithms described by the Advanced Research Projects Agency for the Distributed Interactive Simulation standard. The toolkit also provides treatment of a physically-based class hierarchy for munitions trajectories. In addition, a physically-based, mathematical model for the engines class was implemented.

As a result, a set of DR algorithms have been built to predict the position of simulated entities in all cases. The munitions class implements trajectories for a variety of projectiles. With this arsenal, future versions of NPSNET will be more realistic. The engine class, with new mathematical models, far more realistically represents engine behaviors than the current linear approximation. In summation, the implementation of this toolkit dovetails very well with the needs of NPSNET, and will be integrated into future releases.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

My road to higher education has been long and exhausting. Without the generous support of the following people, I would not have reached this major milestone. I wish to express my gratitude to:

My wife, Cam Ong, who challenged me to forge ahead at those times when my desire flagged.

Dr. Richard Hamming, a friend and great teacher, who enthusiastically taught me how to think and to learn the important things that books alone can never teach.

Dr. David Pratt, my thesis advisor, who lent me guidance and vision when I needed it most.

The U.S. Navy and its representatives who have shown faith in me and given me this opportunity of higher education.

I. INTRODUCTION

A. BACKGROUND

Conventional training and preparation for future military operations is an expensive and time-consuming process involving many persons and resources. With the current budgetary constraints and downsizing of the armed forces, ways of doing business must be revised, but in a way that does not compromise preparedness. Fortunately, with advances in Computer Science, military operations from battle to replenishment can now be simulated on computers, and personnel can complete many phases of training in simulated battlefields, thus mitigating the expense of engaging in the real-world environment.

One such simulated battlefield, NPSNET [PRAT93], has been developed by the Computer Science Department of the Naval Postgraduate School (NPS). NPSNET is a low-cost, distributed, interactive, virtual 3D world for simulation and training. Personnel in the simulated environment interact with each other as if they were on a battlefield. They can virtually fly an airplane, ride a tank, walk, run, and shoot at each other.

NPSNET is an evolving system, a test-bed for new research as well as a growing amalgam of proven ideas. Recently, numerous projects have improved and enhanced NPSNET. Such works encompass: Configuration Management, Real-time Scene Management, Terrain Development, Insertion of the Human, Human-Computer Interface, and Networking. However, NPSNET lacks true physical representation of moving bodies. Only a few simple cases have been implemented such as linear Dead Reckoning (DR), and simplified munitions trajectories for the bullet and bomb. Improving the physically-based aspects of NPSNET is the purpose of this thesis work.

B. PROBLEM STATEMENT

This thesis addresses and resolves three specific deficiencies in the physically-based representations of NPSNET. The first is dead reckoning (DR). NPSNET uses the Distributed Interactive Simulation (DIS) protocol, a standard for networked simulators, to

communicate between entities--aircraft, vehicles, munitions, etc. -- in its environment [IST93]. Every workstation participating in a simulation, each running NPSNET, shares the data for the entities it controls, thus allowing each NPSNET to display a consistent view of the simulated world to its user. Each entity in the environment sends to the network its velocity, position, and orientation no less frequently than every five seconds. Other objects receive and render that information within their own workstation. For efficiency and to conserve network bandwidth, data for an entity is not sent out continually. To keep an updated picture of networked entities, each workstation dead reckons (predicts) remote entities' velocity, position, and orientation based on its last update. Specific dead reckoning algorithms constitute part of the DIS standard. Currently, NPSNET implements only a few simple linear world coordinate cases. A full implementation of DR for all vehicle cases in both world and body coordinates is required.

The second deficiency regards simulation of munitions. In a simulated battle, continuous simulation of munitions is absolutely essential. Currently in NPSNET, only a few simple cases of a bullet and a bomb are implemented. A more definitive physically-based implementation of munitions is required to more closely reflect real-world munitions behavior.

The third deficiency is engine simulation. To make the virtual world realistic and plausible to the user, moving objects must reflect the behavior of their real-world counterparts as much as possible. Currently, NPSNET lacks any engine type. Therefore, a basic implementation of engines is needed, too.

C. APPROACH

For DR implementation, we use the study of the Advanced Research Projects Agency (ARPA) for DR algorithms published as part of DIS 2.0.3 [TOWE94]. In this study, the equations of motion for the DR algorithms are divided into nine groups. The first group is for a static case. Another four groups are for world coordinates, and the remainder are for body coordinates. Using the C++ object-oriented programming methodology, this thesis work implements all nine groups into a class hierarchy.

The munition implementation is based on Newtonian Mechanics. Most moving objects in the real world are governed by gravitational forces. Therefore, kinematic equations with constant downward acceleration are utilized for bullet, bomb, ballistic, and artillery objects. A physically-based model of a system with lost mass is used for rockets and missiles. Air drag is omitted.

For engine implementation, we built an abstract input and output system with general quadratic, logarithmic, and exponential models of acceleration. (A full physically-based model is beyond the scope of this thesis.)

All of the implemented code is written in C++ with class hierarchies for portability and reusability. With its class structure and inheritance, C++ minimizes code development as well as making the program easier to understand.

D. ORGANIZATION

This first chapter introduced NPSNET and discussed the problems addressed by this thesis. It also described the approach taken to solve each problem.The second chapter covers the background and previous works impinging on the problems to be solved. The third chapter describes in detail the implementation of the DR standard. The fourth chapter recounts the formulas and equations which govern the munition classes. The fifth chapter describes the implementation of the engine mathematical models and the derived formulas. The sixth chapter contains the conclusion and presents necessary future work.

II. BACKGROUND AND PREVIOUS WORK

A. PREVIOUS WORKS

1. Dead Reckoning (DR)

Currently, NPSNET implements only a few simple cases of DR where there is no rotation involved. There are the static case, the two cases of world-coordinates, and the default case for the rest of the situations, see Figure 1.

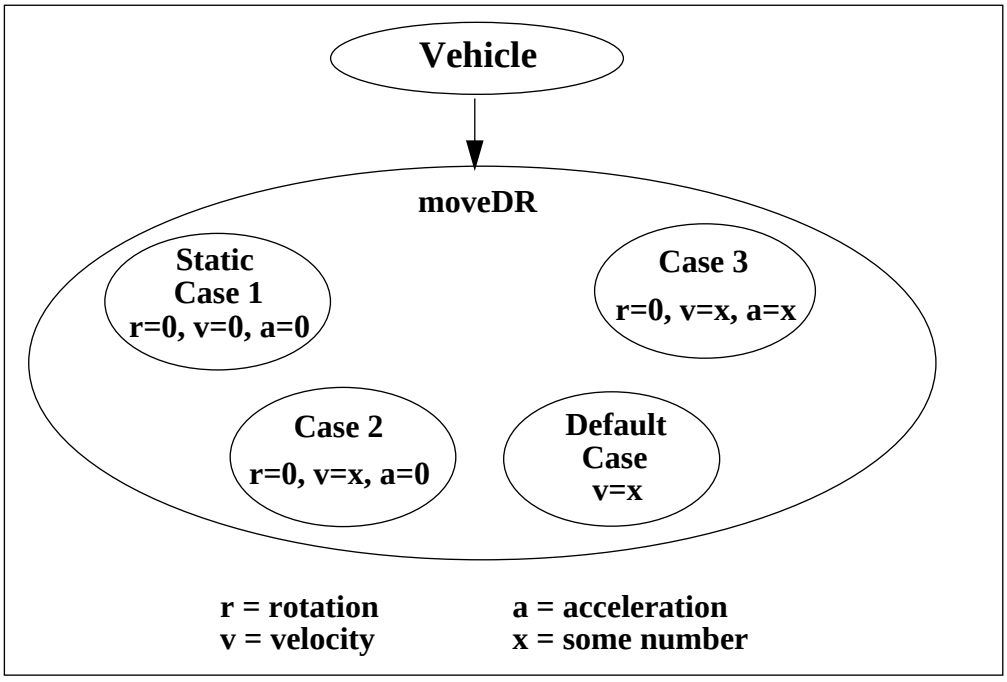

Figure 1: Current NPSNET DR IMPLEMENTATION

The first algorithm applies to a static object when its rotation, velocity, and acceleration are all zero. The second algorithm applies when object rotation and acceleration are zero but velocity is not. The third algorithm applies when object rotation is zero and velocity and acceleration are not. The default case applies when object velocity is a constant, so that the object moves in a straight line. These four DR cases were

implemented in C++ as special cases in the subclass *moveDR* of the Vehicle Class in NPSNET. Kinematic equations with constant acceleration were used to implement those cases:

$$v_i = v_{i_o} + a_i t \qquad \text{Eq 1}$$

$$x_i = x_{i_o} + v_{i_o} t + 0.5 a_i t^2 \qquad \text{Eq 2}$$

Where:

v_i = respective velocity in each direction

v_{i_o} = respective initial velocity in each direction

x_i = respective position in each direction

x_{i_o} = respective initial position in each direction

a_i = respective acceleration of each direction

t = time

One of the purposes of DIS, a set of standards developed by ARPA and industry, is to provide a specification to be used by government agencies and engineers to build interoperable networked simulation systems [IST93]. DIS defined a need for DR, and ARPA released its DR algorithms study on February 7, 1994 [TOWE94]. In this study, with C++ methodology in mind, the algorithms are divided into a general static case and two large groups, one for the world-coordinates, and the other for body-coordinates. Inside each large group are four subgroups, each addressing different cases of the moving object in its respective coordinates, see Figure 2.

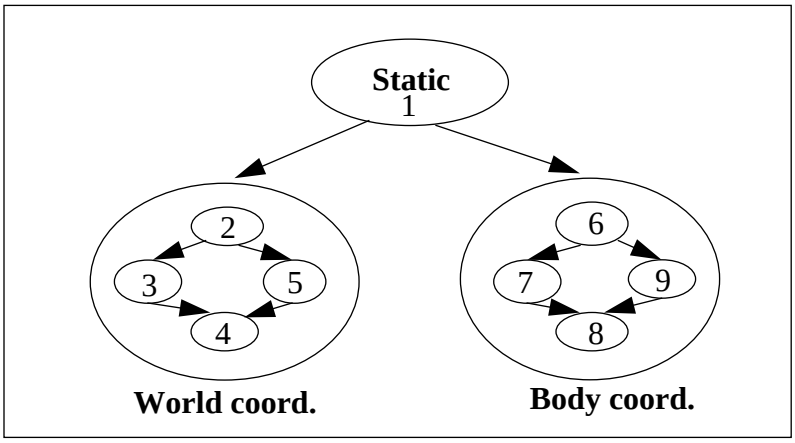

Figure 2: DR algorithms of ARPA's study

2. Munition

In the current version of NPSNET, only a few simple cases of munitions are implemented, such as a bullet, a bomb, and a simple non-loss-mass missile, see Figure 3. Equations similar to Eq1 and Eq2 above are used to implement those munitions.

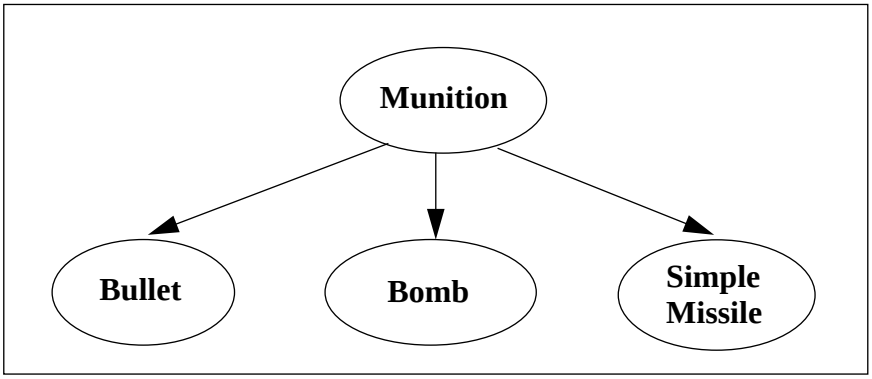

Figure 3: Current NPSNET munitions implementation

Newtonian Mechanics governs the behavior of objects flying in the air in a constant gravitational field. Equations that describe motion and kinematics have been developed ever since Newton presented his greatest discovery (**F = ma**). Many first-year math and physics books describe these equations. [MART84][HALL78]

Another relevant work is the NATO study of ballistic missiles [NATO]. It is the definitive study of the trajectory of a flying mass in the air. It describes all factors influencing the behavior of flying objects, encompassing drag force, lift force, Magnus force, and Coriolis force. Most of these equations exceed the scope of this thesis and are unnecessary for NPSNET, though it could apply to later enhancement work.

3. Engines

Engine models are not currently incorporated into NPSNET. Presently, NPSNET uses a simple linear approximation between the user input to a throttle and the speed of the entity. However, the real world of vehicles encompasses a variety of engine models, such as motors, gas turbine, piston, and rocket engines. All currently existing engine models exceed the scope of this thesis, except for the rocket engine. With a few revisions, it follows the rocket munition model and so is implemented in this thesis work.

B. SUMMARY

In summation, NPSNET lacks complete physically-based models for DR, munitions, and engines, for accurately representing objects in its virtual world. All DR and munition models are readily available, and can be implemented. Except for the rocket engine which can be implemented directly from rocket models, the remaining engine models need to be abstracted as an input and output system and tested until it seems plausible in the virtual world.

III. DEAD RECKONING

A. OVERVIEW

What is dead reckoning? According to the Random House dictionary, dead reckoning is the "calculation of one's position on the basis of distance run on various headings since the last precisely observed position." Our usage of the word dead reckoning in this chapter is exactly the same as the definition.

To allow multiple users in a simulation, we run NPSNET on different workstations which are all connected by the same network. Each workstation presents a consistent view of the entire simulated world, rendering both the local objects it controls and the networked objects controlled by the other simulators. Each simulator transmits the position, velocity, and orientation for its own local objects, and reads the network for information on remote objects. This state change data is sent on the network no less frequently than every five seconds in a packet called, in DIS terminology, a Program Data Unit (PDU). But five seconds is a long time when the user is watching an entity which is animated on the screen. The entity cannot jump to a new location every so many seconds and satisfy the appearance of smooth motion. In-between times, each workstation needs to dead reckon each remote object's position, velocity, and orientation and move it smoothly along its probable course, Figure 4.

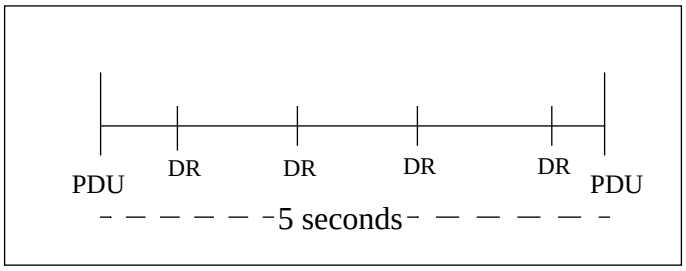

Figure 4: PDU and DR time

In endeavoring to provide a complete set of DR functionality, this thesis utilizes the algorithms developed by ARPA for both world and body coordinates. These are implemented using appropriate C++ classes.

B. DESIGN

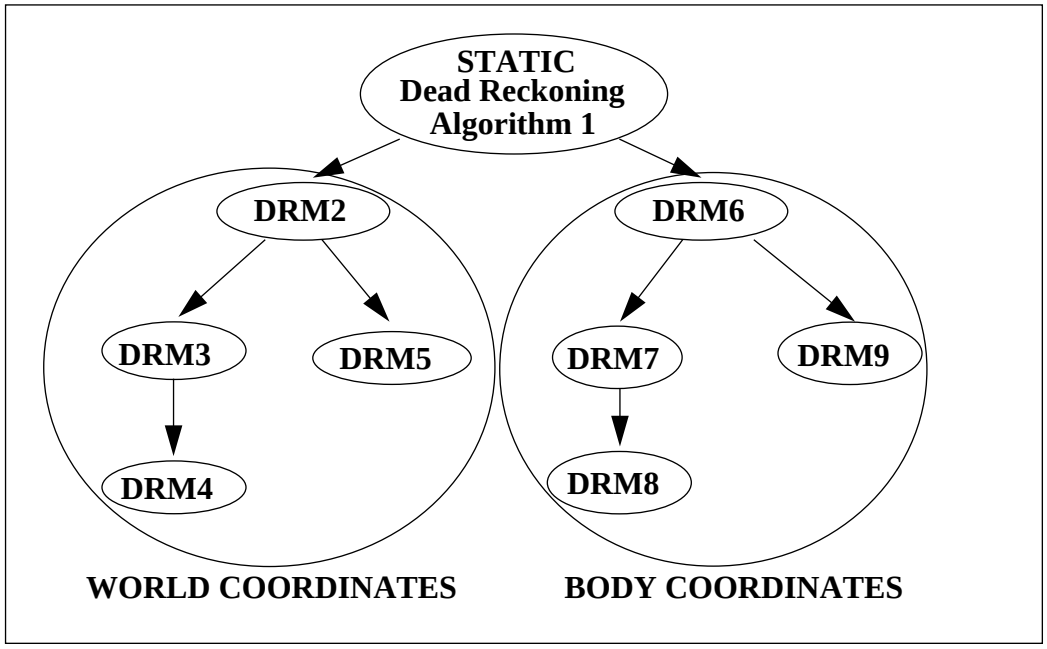

Figure 5: DR classes

In object-oriented languages like C++, *inheritance* is when a child class inherits the variables and functions of a parent class. Thus the child class is automatically like its parent except for the specific ways in which it needs to be different, which makes design and coding clean and efficient. In Figure 5, each of the nine ellipses represents a class of DR algorithm. The parent class is the static class. The four classes on the left side of the structure belong to the world coordinates class, the four on the right belong to the body coordinates.

The structure in Figure 5 varies a bit from ARPA's DR study, see Figure 2. As represented in Figure 2, dead reckoning method four (DRM4) can be derived from either DRM3 or DRM5; likewise for DRM8, which can be derived from DRM7 or DRM9. But, in Figure 5, DRM4 is only derived from DRM3, and DRM8 is only derived from DRM7.

The reason for the variance is to reduce the complications arising from multiple inherited variables and functions. It is far more straightforward to keep track of all the variables and functions if each child class derives from a single parent class.

Table 1 lists each DR algorithm in the set. A key to the abbreviations:

B = body coordinate
F = fixed
P = position
R = rotation
V = velocity
W = word coordinate

For example, the fourth row indicates that algorithm DRM4 has rotation, velocity, and belongs to the world coordinates subclass. Its roll, pitch, yaw (orientation), velocity, and acceleration are equal to some number.

Number	Name	Roll	Pitch	Yaw	Vel	Acc
1	Static	0	0	0	0	0
2	DRM2(F,P,W)	0	0	0	v	0
3	DRM3(R,P,W)	roll	pitch	yaw	v	0
4	DRM4(R,V,W)	roll	pitch	yaw	v	a
5	DRM5(F,V,W)	0	0	0	v	a
6	DRM6(F,P,B)	0	0	0	v	0
7	DRM7(R,P,B)	roll	pitch	yaw	v	0
8	DRM8(R,V,B)	roll	pitch	yaw	v	a
9	DRM9(R,V,B)	0	0	0	v	a

Table 1. DR Algorithms

Furthermore, in the representation DRM(X,Y,Z), X represents for the rotation of the entity, Y represent for position, and Z represent for coordinate.

C. IMPLEMENTATION

1. Static class (DRM1)

This is the parent class for all other DR classes. Its rotation, velocity, and acceleration are all zero, so its position is fixed. To use it, we simply instantiate an object and set its position to given values. In this case, we always get back the same initial value, when we call for updated position.

2. World-coordinates (DRM2-5) classes

The input to all the algorithms are:

$X_w(t_o)$ = position vector in world coordinates at initial time.
$V_w(t_o)$ = velocity vector in world coordinates at initial time.
A_w = acceleration vector in world coordinates.
$(\psi(t_o), \theta(t_o), \phi(t_o))$ = Euler angles at initial time.
$\omega = (\omega_x, \omega_y, \omega_z)^T = (\omega_1, \omega_2, \omega_3)^T$ = angular velocity in body coordinates.
Δt = time increment for dead reckoning step.

The outputs are:

$X_w(t_o + \Delta t)$ = position vector in world coordinates after time delta.
$V_w(t_o + \Delta t)$ = velocity vector in world coordinates after time delta.
$(\psi(t_o + \Delta t), \theta(t_o + \Delta t), \phi(t_o + \Delta t))$ = Euler angles after time delta.

(1) DRM2

In this algorithm, r = 0, v = x, and a = 0. We use the Eq3 and Eq4 to compute the object velocity and position at time $t_o + \Delta t$. Euler angles at time delta $(\psi(t_o + \Delta t), \theta(t_o + \Delta t), \phi(t_o + \Delta t))$ are the same as its initial time $(\psi(t_o), \theta(t_o), \phi(t_o))$ because there is no rotation involved.

$$V_w\left(t_o + \Delta t\right) = V_w\left(t_o\right) + \Delta t A_w \qquad \text{Eq 3}$$

$$X_w\left(t_o + \Delta t\right) = X_w\left(t_o\right) + \Delta t V_w\left(t_o\right) + \frac{1}{2}\Delta t^2 A_w \qquad \text{Eq 4}$$

(2) DRM3

In this algorithm, r = x, v = x, and a = 0. We can get velocity and position at time $t_o + \Delta t$ by Eq3 and Eq4, and the steps to get Euler angles at time $t_o + \Delta t$ are as follows:

Step 1: Use the initial Euler angles $(\psi(t_o), \theta(t_o), \phi(t_o))$ to compute the rotations matrix $U(b(t_o) \mid w)$ which takes world coordinates into body coordinates at time t_o. The formula for $U(b(t_o) \mid w)$ is

$$U\left(b\left(t_o\right) \mid w\right) = \begin{bmatrix} c(\psi)\,c(\theta) & s(\psi)\,c(\theta) & -s(\theta) \\ -s(\psi)\,c(\phi) + s(\phi)\,s(\theta)\,c(\psi) & c(\phi)\,c(\psi) + s(\phi)\,s(\theta)\,s(\psi) & s(\phi)\,c(\theta) \\ s(\phi)\,s(\psi) + c(\phi)\,s(\theta)\,c(\psi) & -s(\phi)\,c(\psi) + c(\phi)\,c(\theta)\,s(\psi) & c(\phi)\,c(\theta) \end{bmatrix} \qquad \text{Eq 5}$$

Where c = cosine and s = sine.

Step 2: Compute the rotation matrix $U(b(t_o + \Delta t) \mid b(t_o))$ which takes body coordinates at time t_o into body coordinates at time $t_o + \Delta t$.

$$U\left(b\left(t_o + \Delta t\right) \mid b\left(t_o\right)\right) = \frac{(1 - \cos(|\omega|\Delta t))}{|\omega|^2}\omega\omega^T + \cos(|\omega|\Delta t)\, I - \frac{\sin(|\omega|\Delta t)}{|\omega|}\Omega \qquad \text{Eq 6}$$

Where:

$$\Omega = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \qquad \text{Eq 7}$$

The matrix I is the 3x3 identity matrix having 1's on the main diagonal and 0's off of the main diagonal. The matrix $\omega\omega^T$ is

$$\omega\omega^T = \begin{bmatrix} \omega_1\omega_1 & \omega_1\omega_2 & \omega_1\omega_3 \\ \omega_2\omega_1 & \omega_2\omega_2 & \omega_2\omega_3 \\ \omega_3\omega_1 & \omega_3\omega_2 & \omega_3\omega_3 \end{bmatrix} \quad \text{Eq 8}$$

and the magnitude of ω is

$$|\omega| = (\omega_1^2 + \omega_2^2 + \omega_3^2)^{1/2} \quad \text{Eq 9}$$

Step 3: Compute the rotation matrix $U(b(t_o + \Delta t)|w)$ which takes world coordinates into body coordinates at time $t_o + \Delta t$.

$$U(b(t_o + \Delta t)|w) = U(b(t_o + \Delta t)|b(t_o)) \cdot U(b(t_o)|w) \quad \text{Eq 10}$$

Step 4: Compute the Euler angles $(\psi t_o + \Delta t), \theta(t_o + \Delta t), \phi(t_o + \Delta t))$ after the time increment using the rotation matrix $U(b(t_o + \Delta t)|w)$. The rotation matrix $U(b(t_o + \Delta t)|w)$ has the form:

$$U(b(t_o + \Delta t)|w) = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ U_{21} & U_{22} & U_{23} \\ U_{31} & U_{32} & U_{33} \end{bmatrix} \quad \text{Eq 11}$$

Where the entries U_{ij} are functions of the Euler angles $(\psi(t_o + \Delta t), \theta(t_o + \Delta t), \phi(t_o + \Delta t))$ as shown in the matrix displayed in Step 3 above.

Step 5: The Euler angle θ is recovered via

$$\theta = -\text{asin}(U_{13}) \in \left[\frac{-\pi}{2}, \frac{\pi}{2}\right] \quad \text{Eq 12}$$

If U_{11} and U_{12} do not both equal zero, the Euler angle ψ is recovered via

$$\psi = Arg(U_{11}, U_{12}) \in [-\pi, \pi] \quad \text{Eq 13}$$

Where the function Arg(x,y) is an argument of the complex number x + yi, the particular branch of the argument chosen to lie in the interval $[-\pi,\pi]$.

If U_{23} and U_{33} do not both vanish, the Euler angle ϕ is recovered via

$$\phi = Arg(U_{33},U_{23}) \qquad \text{Eq 14}$$

The exceptional cases occur when $\theta = \pi/2$ or $-\pi/2$. In that case, the Euler angles ψ and ϕ are recovered via

$$\psi = 0 \qquad \text{Eq 15}$$

$$\phi = Arg(U_{22},-U_{32}) \qquad \text{Eq 16}$$

(1) DRM4

In this algorithm, r = x, v = x, and a = x. Position, velocity, and orientation are computed much the same as DRM3 but with the addition of acceleration A_w.

(3) DRM5

In this algorithm, r = 0, v = x, and a = x. We can use Eq3 and Eq4 above to compute its position and velocity. DRM5 lacks rotation so we do not require equations Eq5 through Eq16. Its orientation is fixed.

3. Body-coordinates (DRM6-9) classes

The input to all the algorithms are:

$X_w(t_o)$ = position vector in world coordinates at initial time.

$V_b(t_o)$ = velocity vector in body coordinates at initial time.

A_b = time derivative of V_b

$(\psi(t_o),\theta(t_o),\phi(t_o))$ = Euler angles at initial time.

$\omega = \left(\omega_x,\omega_y,\omega_z\right)^T = \left(\omega_1,\omega_2,\omega_3\right)^T$ = angular velocity in body coordinates.

Δt = time increment for dead reckoning step.

The outputs are:

$X_w\left(t_o + \Delta t\right)$ = position vector in world coordinates after time delta.

$V_b\left(t_o + \Delta t\right)$ = velocity vector in body coordinates after time delta.

$\left(\psi\left(t_o + \Delta t\right), \theta\left(t_o + \Delta t\right), \phi\left(t_o + \Delta t\right)\right)$ = Euler angles after time delta.

(4) DRM6

In this algorithm, r = 0, v = x, and a = 0. We can compute the velocity and position at $t_o + \Delta t$ with Eq17 and 18, ignoring second terms on the right of both equations because they are zero. The Euler angles at time $t_o + \Delta t$ is the same as its initial condition $(\psi(t_o), \theta(t_o), \phi(t_o))$, because there is no rotation involved.

$$V_b\left(t_o + \Delta t\right) = V_b\left(t_o\right) + \Delta t A_b \qquad \text{Eq 17}$$

$$X_w\left(t_o + \Delta t\right) = X_w\left(t_o\right) + U\left(w|b\left(t_o\right)\right)\left(\int_0^{\Delta t} e^{\tau\Omega} d\tau V_b\left(t_o\right) + \int_0^{\Delta t} \tau e^{\tau\Omega} d\tau A_b\right) \qquad \text{Eq 18}$$

$$\int_0^{\Delta t} e^{\tau\Omega} d\tau = \frac{|\omega|\Delta t - \sin(|\omega|\Delta t)}{|\omega|^3}\omega\omega^T + \frac{\sin(|\omega|\Delta t)}{|\omega|}I + \frac{1 - \cos(|\omega|\Delta t)}{|\omega|^2}\Omega \qquad \text{Eq 19}$$

$$\int_0^{\Delta t} \tau e^{\tau\Omega} d\tau = \frac{\frac{1}{2}|\omega|^2\Delta t^2 - \cos(|\omega|\Delta t) - |\omega|\Delta t\sin(|\omega|\Delta t) + 1}{|\omega|^4}\omega\omega^T +$$

$$\frac{\cos(|\omega|\Delta t) + |\omega|\Delta t\sin(|\omega|\Delta t) - 1}{|\omega|^2}I + \frac{\sin(|\omega|\Delta t) - |\omega|\Delta t\cos(|\omega|\Delta t)}{|\omega|^3}\Omega \qquad \text{Eq 20}$$

Computing $|\omega|$ and Ω is the same as in Eq9 and Eq7, and $U\left(w|b\left(t_o\right)\right)$ is the transposition of the matrix $U(b(t_o)|w)$ above. See Eq5.

The following steps are used to calculate Euler angles $(\psi(t_o + \Delta t), \theta(t_o + \Delta t), \phi(t_o + \Delta t))$ at time $t_o + \Delta t$.

Step 1: Compute the rotation matrix $U(b(t_o + \Delta t) \mid (t_o))$ which takes body coordinates at time t_o into body coordinates at time $t_o + \Delta t$, the same as Eq 6.

Step 2: Compute the rotation matrix $U(b(t_o + \Delta t) \mid w)$ which takes world coordinates into body coordinates at time $t_o + \Delta t$ as Eq 10.

Step 3: Compute the Euler angles $(\psi(t_o + \Delta t), \theta(t_o + \Delta t), \phi(t_o + \Delta t))$ after the time increment using the rotation matrix $U(b(t_o + \Delta t) \mid w)$, same as the Step 4 in the world coordinates.

(5) DRM7

In this algorithm, r = x, v = x, a = 0. We can compute the velocity, position, and Euler angles at $t_o + \Delta t$ with Eq3, Eq4, and Steps 1 to 3.

(1) DRM8

The steps to compute the outputs for this algorithm, where r = x, v = x, and a =x, are the same as the steps of DRM7 with the exception that a is not zero.

(6) DRM9

In this algorithm, r = 0, v =x, and a =x. The steps to get the outputs are the same as DRM6 with a not equal to that zero. The only thing changes in this algorithm is that the second term on the right of Eq17 is not zero.

D. RESULTS

With the organization afforded by the C++ class hierarchy, the coding and testing procedures for DR classes are segregated into separate, discrete units. The debugging process for each implemented algorithm was very simple. For each case, the code was tested, as suggested in Figure 4, as follows (see code in Appendix A for details):

a. Instantiate an object of that case.

b. Call procedure *update_DR* to set the object to its "real" values (as the information is obtained from incoming PDUs).

c. Intermittently call *move_DR* to compute the predicted position, velocity, and angular position of the object.

d. Repeat step c. at the desired interval until the next PDU arrives with real data for the object, then repeat step b.

The code is extremely reliable, and its performance very quick, even when tested on a low-end Intel 80486-based PC. To illustrate test results, the following Tables 2 and 3 list the input and output tested cases for algorithms DRM4 and DRM8. They are the two most complex cases in the DR classes.

Initial Conditions	x	y	z
Position - m	-6378137	-5	-10
Velocity- m/sec	30	35	40
Acceleration - m/sec*sec	-80	-85	-90
Orientation - deg	15	20	25
Rotation Rate - deg/sec	60	65	70
Final Results			
Position - m	-6378132.00	1.88	-1.25
Velocity - m/sec	-10.00	-7.50	-5.00
Orientation - deg	64.32	14.84	72.50

Table 2. Test of DR algorithm 4, Delta time is 0.5 sec

Initial conditions	x	y	z
Position - m	6378137	5	10
Velocity - m/sec	30	35	40
Acceleration - m/sec*sec	-80	-85	-90
Orientation - deg	15	20	25
Rotation Rate - deg/sec	-60	-65	-70
Final Results			
Position - m	6378144.03	10.46	18.32
Velocity - m/sec	-10.00	-7.50	-5.00
Orientation - deg	-23.07	-8.68	-10.86

Table 3. Test of DR algorithm 8, delta time is 0.5 sec

Following Table 4 is a time performance data comparison between this thesis's code implementation in C++ and ARPA's study code implementation in C. Those implementations were run on different platforms to test the consistency and time usage of the code.

DRM	ARPA's study SGI Indigo 2 100MHz	ARPA's study SGI Indigo 2 150MHz	Thesis Implementation SGI Indigo 2 150MHz	Thesis Implementation SGI Indigo 2 150MHz Optimized	Thesis Implementation 486DLC 33MHz
1	1.9 usec	1.3 usec	0.08 usec	0.08 usec	7 usec
2	3.8 usec	2.6 usec	0.96 usec	0.40 usec	45 usec
3	55 usec	30 usec	35.20 usec	26.32 usec	900 usec
4	57 usec	30 usec	36.64 usec	26.69 usec	980 usec
5	5.9 usec	4.0 usec	2.08 usec	1.12 usec	7 usec
6	20 usec	11 usec	0.24 usec	0.16 usec	20 usec
7	63 usec	33 usec	51.60 usec	34.56 usec	1400 usec
8	71 usec	38 usec	56.84 usec	44.20 usec	1900 usec
9	22 usec	13 usec	0.88 usec	0.40 usec	50 usec

Table 4. Performance Data

Observe that the trend of the times spent on computation were similar in all cases and different platforms. Without optimization in compiling, this thesis implementation ran fast in the simple cases of algorithms 1, 2, 5, 6, and 9, but slow in the complicated cases of 3, 4, 7, and 8. Using compilation optimization (with the flag "-O3 -mips2"), and considering the lower overhead resulting from putting each class into its own file, most of the cases ran faster than the ARPA's study code. However, in the complex case of algorithm 8, our code ran a little slower due to redundant matrix multiplication in computing angular position. The speed of the 486DLC 33Mhz CPU is about three times slower than the Intel 486DX2 66MHz CPU.

In summation, the code for each of the DR algorithms works as designed, and has been implemented in the current NPSNET. NPSNET thus takes an important step forward in DIS-compliance. NPSNET can move networked entities from disparate simulators following any of the DR algorithms those entities specify.

IV. MUNITIONS

A. OVERVIEW

One of the goals of simulation is to make the virtual world resemble the real world as closely as possible. This chapter describes an extensive munitions class, built so that NPSNET programmers can enhance and improve the physically-based representation of munitions in NPSNET. The class incorporates the mathematics that will allow various munitions to follow realistic trajectories in the simulated world.

The kinematic equations $v = v_o + at$ and $x = x_o + v_o t + 0.5at^2$ calculate velocity and position for most of the munitions, thus forming the core of the base class. Orientation of the munition is derived from its current velocity vector. Implementation of the subclasses of the base class is straightforward from those equations, except for the rocket model which involves a second order differential equation.

B. DESIGN

The munition class is the base class containing all the common variables and functions for its subclasses, see Figure 6. The equations that govern the velocity and position of the bullet, bomb, and ballistic subclasses are much the same. In the artillery class, though, we find the artillery's necessary initial velocity first, then it uses the same equation as its parent class--bomb/ballistic--to find its velocity and position at a later time. The rocket subclass is totally different from its peers. Its velocity and position are governed by a different set of complex equations. The approach to compute the velocity and position of the ballistic missile class is the same as for the artillery class. That is, its initial conditions are determined before using the rocket's velocity and position functions to find its velocity and position at a later time.

The guided missile seeks and destroys its target. It is a subclass of the base class. Incorporated with the engine class, its trajectory can be further refined by factoring in the power curve as it is affected by user input. As examples, in one case the missile may couple

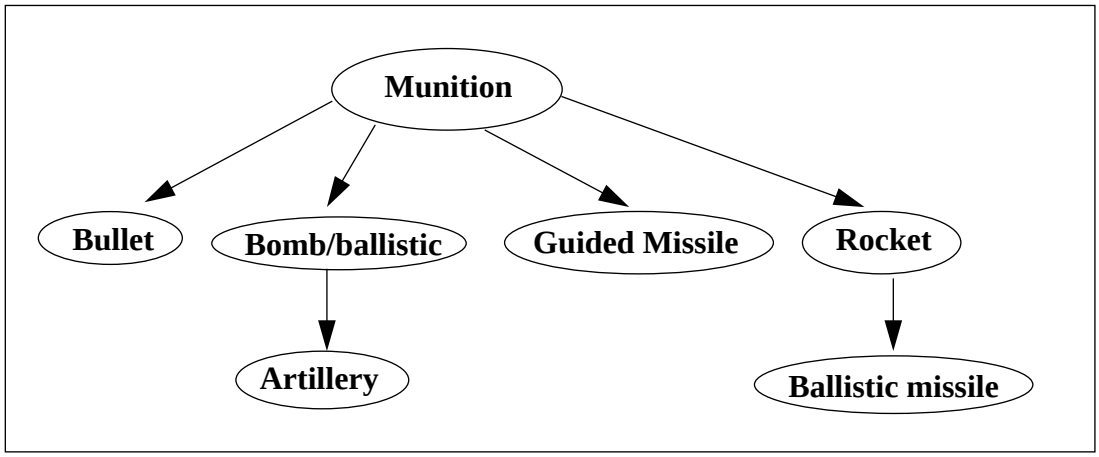

Figure 6: Munition class structure

with a constant speed curve to chase after the target, while in another it couples with the quadratic engine speed equation.

C. IMPLEMENTATION

1. Orientation

The orientation of the munition is converted from its velocity vector, a function residing in the munition base class. The function takes a velocity vector and returns angular position in degrees, the values commonly used by graphics software. This function checks all eight cases of given (x, y, z) values, and then used inverse tangent or direct assignment to assign appropriate angular position for each direction.

For example, if the input vector is (x, y, z) then:

$$\theta_x = \text{r\_to\_d} \cdot \text{atan}\frac{x}{y} \qquad \text{Eq 21}$$

$$\theta_y = \text{r\_to\_d} \cdot \text{atan}\frac{y}{x} \qquad \text{Eq 22}$$

$$\theta_z = \text{r\_to\_d} \cdot \text{atan}\frac{z}{x} \qquad \text{Eq 23}$$

Where r_to_d is the factor to change from radian to degrees. If the input vector is (0, y, z) then:

$$\theta_x = 90 \quad \text{Eq 24}$$

$$\theta_y = \text{r\_to\_d} \cdot \text{atan}\frac{y}{z} \quad \text{Eq 25}$$

$$\theta_z = \text{r\_to\_d} \cdot \text{atan}\frac{z}{y} \quad \text{Eq 26}$$

2. Bullet

A bullet is a small projectile fired from small arms. To simulate its path, we use equations Eq 27 through Eq 32. The gravity constant is set to one or zero to implement the bullet with or without the effect of gravity, see Figure 7 and equations Eq 27 through Eq 32.

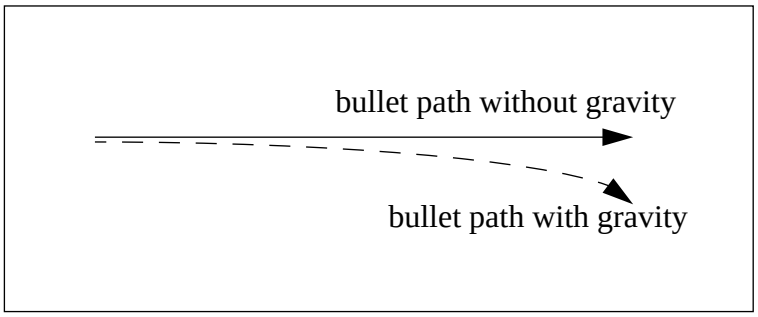

Figure 7: Bullet path

Inputs:

v_{i_o} initial velocity in the respective direction.

x_{i_o} initial position in the respective direction.

Outputs:

$v_i(t)$ velocity at time t in the respective direction.

$x_i(t)$ position at time t in the respective direction.

$\omega_i(t)$ angular position at time t in the respective direction.

The following equations Eq 27 through Eq 29 describe the velocity of the bullet. Velocity in the x and y axes of the horizontal plane will not change from the initial state

because there is no significant force acting on it. We omit the air drag and wind effects, which are negligible. Gravitational force acts upon the z direction.

$$v_x = v_{x_o} \quad \text{Eq 27}$$

$$v_y = v_{y_o} \quad \text{Eq 28}$$

$$v_z = v_{z_o} - gt \cdot gravity \quad \text{Eq 29}$$

The following equations Eq 30 through Eq 32 describe the position of the bullet. Position, at any time, is based on initial position and initial velocity. In the z direction, the bullet drops due to gravitational force.

$$x_x = x_{x_o} + v_{x_o} t \quad \text{Eq 30}$$

$$x_y = x_{y_o} + v_{y_o} t \quad \text{Eq 31}$$

$$x_z = x_{z_o} + v_{z_o} t - 0.5gt^2 \cdot gravity \quad \text{Eq 32}$$

3. Bomb

Typically, a bomb is dropped from a flying object. Due to gravity, it descends from its initial position and accelerates toward the surface of the earth. Its path, normally, is a parabola, Figure 8. The equations governing its velocity and position are much the same as the equations for the bullet, Eq 27 through Eq 32. Inputs and outputs are the same as for the bullet class above.

Together with Eq 21 and Eq 22, the following equation describes the velocity of the bomb.

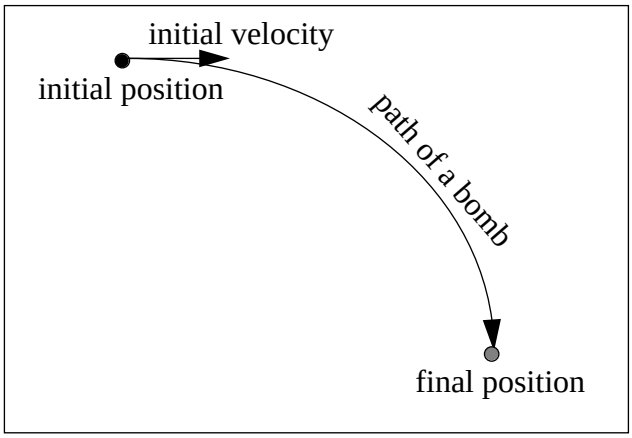

Figure 8: Path of a bomb

$$v_z = v_{z_o} - gt \qquad \text{Eq 33}$$

Together with Eq 30and Eq 31, the following equation describes the position of the bomb.

$$x_z = x_{z_o} + v_{z_o} t - 0.5gt^2 \qquad \text{Eq 34}$$

These equations are the same as Eq 27 through Eq 32. The difference is the "*gravity*" constant of Eq 29 and Eq 32 is omitted, since bombs are always dropped from high above the surface where gravitational force always applies.

4. Ballistic

Ballistic behavior applies to projectiles, like an artillery round, which are shot not directly at the target, but along a parabola whose path is calculated to culminate at the target location, Figure 9. Inputs and outputs are the same as for the bullet and bomb classes above.

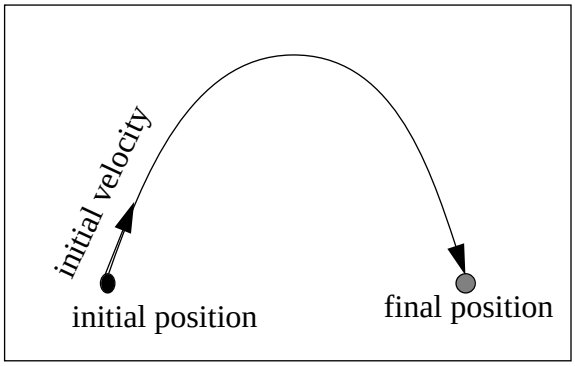

Figure 9: Path of the ballistic

Equations that govern the behavior of ballistic projectiles are much the same as equations for the bomb. The only difference between the two classes are the initial conditions, that is, the ballistic projectile normally originates from the ground and the bomb from the air.

a. Artillery

The path of the artillery round is the same as for the general ballistic round. The difference is in the given conditions. For the ballistic, we are given the initial velocity, and for the artillery we are given the artillery and the target positions.

Inputs:

P1(x1, y1, z1) initial position of the artillery.

P2(x2, y2, z2) initial position of the target.

Outputs: the same as for the bullet.

In the following case, we only know two positions: P1 and P2. We do not know the artillery round's initial velocity, see Figure 8. If we did know the artillery's initial velocity, the way to find the position at time t, P(t), would be the same as for the bomb or ballistic. So the approach is to find the initial velocity first. Once we have the initial condition, the equations that govern the behavior of the velocity and the position at a later time are the same as the equations for the bomb or ballistic.

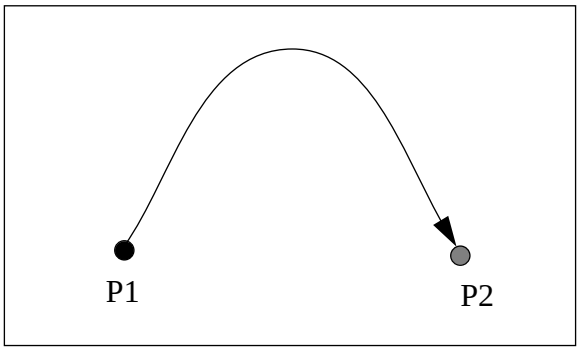

Figure 10: Artillery's round path

The following equations Eq 35 through Eq 37 are used to find the required nozzle velocity of the artillery round.

$$v_{x_O} = \frac{x - x_o}{t} \quad \text{Eq 35}$$

$$v_{y_o} = \frac{y - y_o}{t} \quad \text{Eq 36}$$

$$v_{z_o} = \frac{\left| z - z_o + 0.5gt^2 \right|}{t} \quad \text{Eq 37}$$

In Eq 35 through Eq 37, the unknown variable on the right is the time, t, so the programmer can code the implementation to query the user for that value. In Eq 37, the top right part takes the absolute value to ensure that the velocity in the z direction is positive (going up).

5. Rocket

A rocket is a type of munition that burns fuel, shooting it rearwards to gain velocity and altitude, hence distance (Figure 11). At a certain burn-out point, the rocket will have expended its fuel. From that point, it enters free-fall as a ballistic projectile. Its subsequent path is based on the velocity and position gained in the burn phase. Therefore, there are two

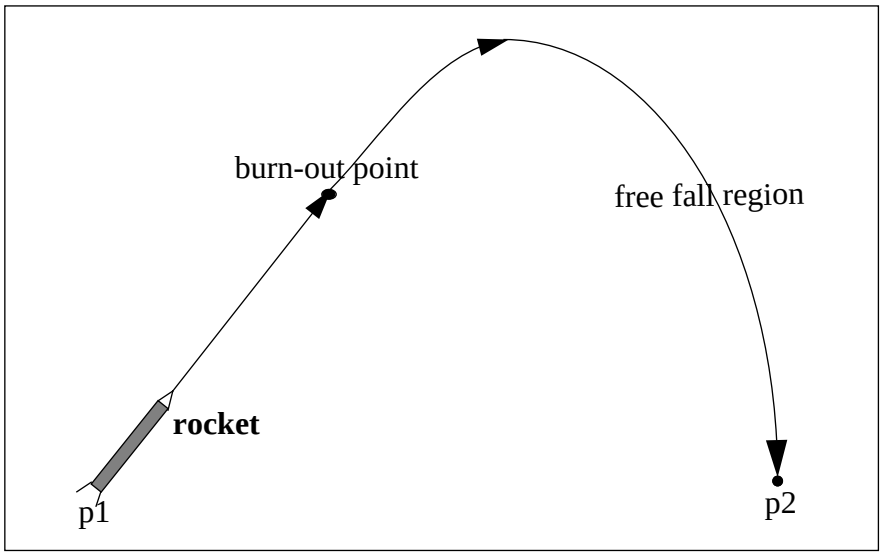

Figure 11: Rocket's path

set of equations governing the rocket's velocity and position, the first applying to the burn phase, the second to the free-fall phase.

Inputs: all the variables on the right side of the following velocity and position equations.

Outputs: the same as before: velocity, position, and angular position.

Equation 38 states that the rate of change of the momentum (P = mv) is equal to the sum of the external forces that act on the object.

$$\frac{d}{dt}\vec{P} = \sum \vec{F}_{ext} \qquad \text{Eq 38}$$

Based on Eq 38, we can derive the equation for the motion of an object with lost mass, Eq 39, where m is mass, V is velocity, u is the velocity of the fuel, t is time, and F is the external force. Eq 39 states that mass multiplied by the rate of change of velocity, minus velocity multiplied by the rate of change of mass, equals the external force that applies to the object.

$$m\frac{d}{dt}\vec{V} - \vec{u}\frac{dm}{dt} = \vec{F}_{ext} \qquad \text{Eq 39}$$

From Eq 39, and if we assume that $\frac{dm}{dt} = r$ where r, rate of lost mass, is a constant, we can derive the velocity and position equations. The derivation can be found in many math and physics books [MART84][HALL78][BOYC77].

Eq 40 through Eq 42 following are the velocity equations for the rocket.

$$v_z - v_{z_o} = u_z \ln\left(1 - \frac{rt}{m_o}\right) - gt \qquad \text{Eq 40}$$

$$v_x - v_{x_o} = u_x \ln\left(1 - \frac{rt}{m_o}\right) \qquad \text{Eq 41}$$

$$v_y - v_{y_o} = u_y \ln\left(1 - \frac{rt}{m_o}\right) \qquad \text{Eq 42}$$

Eq 43 through Eq 45 following are the equations for the position of the rocket.

$$x_z - x_{z_o} = v_{z_o} t - \frac{u_z m_o}{r}\left\{1 - \left(1 - \frac{rt}{m_o}\right)\left[1 - \ln\left(1 - \frac{rt}{m_o}\right)\right]\right\} - \frac{1}{2}gt^2 \qquad \text{Eq 43}$$

$$x_x - x_{x_o} = v_{x_o} t - \frac{u_x m_o}{r}\left\{1 - \left(1 - \frac{rt}{m_o}\right)\left[1 - \ln\left(1 - \frac{rt}{m_o}\right)\right]\right\} \qquad \text{Eq 44}$$

$$x_y - x_{y_o} = v_{y_o} t - \frac{u_y m_o}{r}\left\{1 - \left(1 - \frac{rt}{m_o}\right)\left[1 - \ln\left(1 - \frac{rt}{m_o}\right)\right]\right\} \qquad \text{Eq 45}$$

Where:

$\vec{v}_i$ = respective velocity in each direction

$\vec{x}_i$ = respective position in each direction

$\vec{u}_i$ = respective fuel velocity in each direction

m_o = total mass of the rocket

r = rate of lost mass

g = gravitational constant, 9.81m/s^2

$\dot{v}_{i_o}$, $\dot{x}_{i_o}$ = respective initial velocity and position

t = time

If we assume that fuel mass will burn-out at a certain time, as in Eq 46, we can derive the burn-out velocity and the burn-out position as in equations Eq 47 through Eq 52.

$$m_{fuel} = m_o - m_{rocket} = rt \qquad \text{Eq 46}$$

Where:

m_{fuel} = fuel mass

m_{rocket} = mass of the bare rocket

The following equations Eq 47 through Eq 49 give the burn-out velocity of the rocket.

$$v_z - v_{z_o} = -u_z \ln\left(1 + \frac{m_f}{m_r}\right) - g\frac{m_f}{r} \qquad \text{Eq 47}$$

$$v_x - v_{x_o} = -u_x \ln\left(1 + \frac{m_f}{m_r}\right) \qquad \text{Eq 48}$$

$$v_y - v_{y_o} = -u_y \ln\left(1 + \frac{m_f}{m_r}\right) \qquad \text{Eq 49}$$

The following equations Eq 50 through Eq 52 give the burn-out position of the rocket.

$$x_z - x_{z_o} = v_{z_o} t - \frac{u_z m_o}{r}\left\{1 - \left(\frac{m_r}{m_o}\right)\left[1 - \ln\left(1 + \frac{m_f}{m_r}\right)\right]\right\} - \frac{1}{2}g\left(\frac{m_f}{r}\right)^2 \quad \text{Eq 50}$$

$$x_x - x_{x_o} = v_{x_o} t - \frac{u_x m_o}{r}\left\{1 - \left(\frac{m_r}{m_o}\right)\left[1 - \ln\left(1 + \frac{m_f}{m_r}\right)\right]\right\} \quad \text{Eq 51}$$

$$x_y - x_{y_o} = v_{y_o} t - \frac{u_y m_o}{r}\left\{1 - \left(\frac{m_r}{m_o}\right)\left[1 - \ln\left(1 + \frac{m_f}{m_r}\right)\right]\right\} \quad \text{Eq 52}$$

a. Ballistic missile

The inter-continental ballistic missile (ICBM) is a type of weapon which uses rocket thrust and has the capability of flying directly to a given target position, Figure 12. The equations governing its velocity and position are the same as for the artillery class. We determine the ICBM initial conditions. Then we use the parent class velocity and position equations to find the velocity and position at a later time.

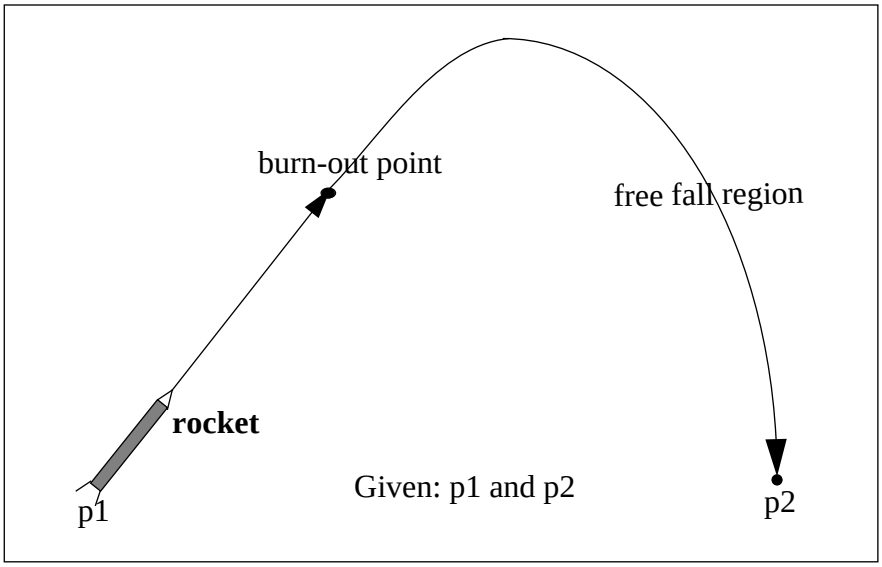

Figure 12: Rocket shot at predetermined target.

6. Guided missile

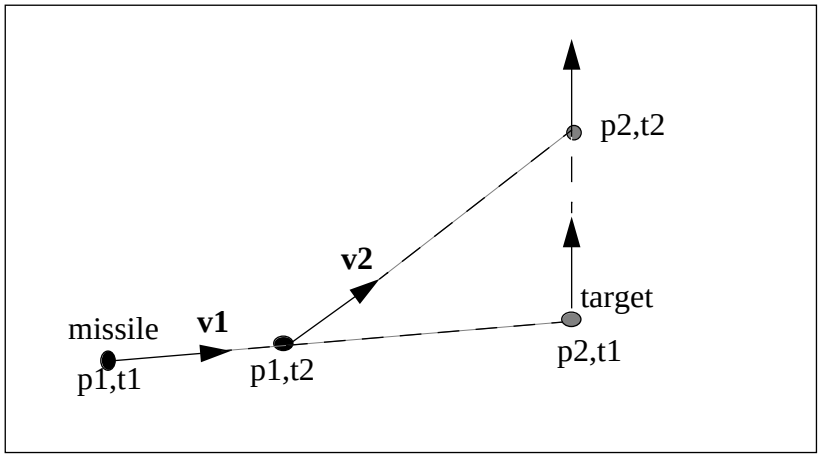

Figure 13: Missile chasing target

A guided missile is a projectile with the capability of chasing and homing in on a moving target, Figure 13. Our general solution for the moving target will apply to the fixed target case, too. In the moving target case, the missile must change its velocity vector every delta time, Δt = t2 - t1. At the end of each delta time, we know the position and velocity of both the missile and target. We use that as the initial conditions for the next time interval. We repeat the process until the missile is within a detonation range from the target.

The following equation Eq 53 computes the new missile velocity for every Δt.

$$\hat{v}_{missile} = \frac{\hat{P}_{target} - \hat{P}_{missile}}{\left\| \hat{P}_{target} - \hat{P}_{missile} \right\|} Speed \qquad \text{Eq 53}$$

Where:

$\hat{P}_{target}$ = target position

$\vec{v}_{missile}$ = new missile velocity

$\vec{P}_{missile}$ = missile position

$Speed$ = speed of the missile from previous delta time

Following is the position equation for the guided missile

$$x_i = x_{i_o} + v_{i_o}\Delta t + 0.5a_i(\Delta t)^2 \qquad \text{Eq 54}$$

Where i = is the respective direction, x_i = next position, x_{i_o} = current position direction, v_{i_o} = velocity at current position, a_i = acceleration, and Δt = given delta time. Remember that v_{i_o} is recalculated every Δt.

D. RESULTS

As in Chapter 3, each case of the munitions class is coded and tested separately. The code for each case resides in separate header and body files. This divide-and-conquer strategy facilitated the coding and testing process tremendously. Potential problems and errors were contained within sections of code of manageable size. Once the problems of the parent class are ironed out, those parent class attributes inherited by the child classes need not be validated and debugged again for each child class. With the C++ hierarchy structure, problems are isolated and localized, and not propagated.

The following two pseudo-code procedures outline the testing of the munitions subclasses, and the guided missile class which is handled separately:

Following is the main function used to test the munition subclasses:

----------begin main procedure----------

1. Instantiate an object
2. While object does not hit the target, or the ground
    a. Move the object
    b. Get the object position (for rendering)
    c. Get the object velocity

d. Get the object angular position (for rendering)
e. Check whether the object has hit the target or ground
(if it hits, exit the loop; if not, continue)
f. Check whether the object has missed the target
(if so, exit the loop)
--------- end while loop----------

----------end main procedure----------

The following loop describes the coding and testing of the guided missile class:

----------begin main procedure----------

1. Set up initial conditions (delta time, missile position, target position)
2. While the target has not been hit
    a. Compute missile velocity, angular position
    b. Move the missile for a given delta time
    c. Get the next target position (from DR)
    d. Get the next speed (constant or from speed curve)
    e. Determine if the missile hit the target within a given range
    (if so, exit the loop)
3. Return to Step 2

----------end main procedure----------

Note: If the delta time is not small enough and the detonated range not large enough, the definition of a hit required by step e will never be satisfied.

As the two pseudo-code procedures above describe, the munition's state is started with initial conditions and then recalculated after a predetermined small delta time, with checks of whether the target has been hit or the munition has impacted the ground. If one of those conditions is true, the loop is exited and the procedure terminated.

In summation, the code works as designed. Each class returns its output in a well-behaved manner. The code is reliable, and the performance excellent, even on the low-end Intel 80486-based PC used for testing.

V. ENGINES

A. OVERVIEW

This chapter describes mathematical models for engine power. The actual implementation of engine models such as the gas turbine, motor, and rocket are not essential to the virtual world of NPSNET. However, it is important that the simulation looks and feels right to the user. Therefore, our goal was to design our equations only to a level of detail that will have a beneficial effect on the simulation.

Figure 14 shows all of the mathematical models which will be discussed in this chapter. Depending on the model we use, engine speed will follow the mathematical curve precisely, either the logarithmic, sinusoidal, linear, exponential, rocket, or quadratic curve.

Note that we are only concerned with increasing or decreasing speed. The direction of any object employing our engine model depends on user input during the simulation.

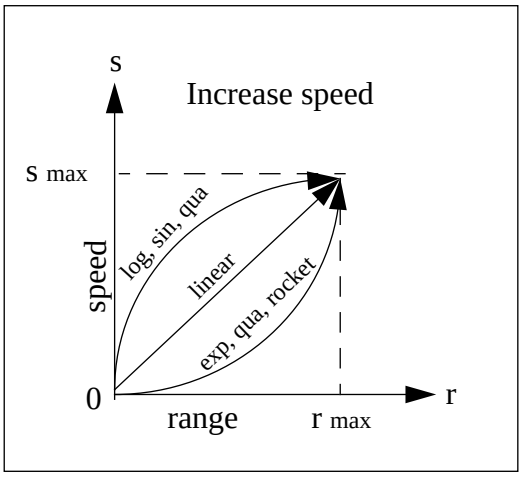

Figure 14: Models for Engines

In Figure 14, s represents the speed of the entity and r is the range of the input throttle. Variable r represents, for instance, pressing the gas pedal in a car. As r increases, so does the speed of the car and vice versa. v_{max} is the maximum speed that the object can

attain, and r_{max} is the maximum range number that the user will supply. For example, r might run from 0 to 100.

B. DESIGN

The engine class is the base class. It contains the common variables the subclasses, which are derived from the base class, will use. Most subclasses have two cases implemented, one each for increasing and decreasing speed. I and D in Figure 15 shows that all subclasses implement both cases, except for the rocket, which will not follow the same curve for decreasing speed.

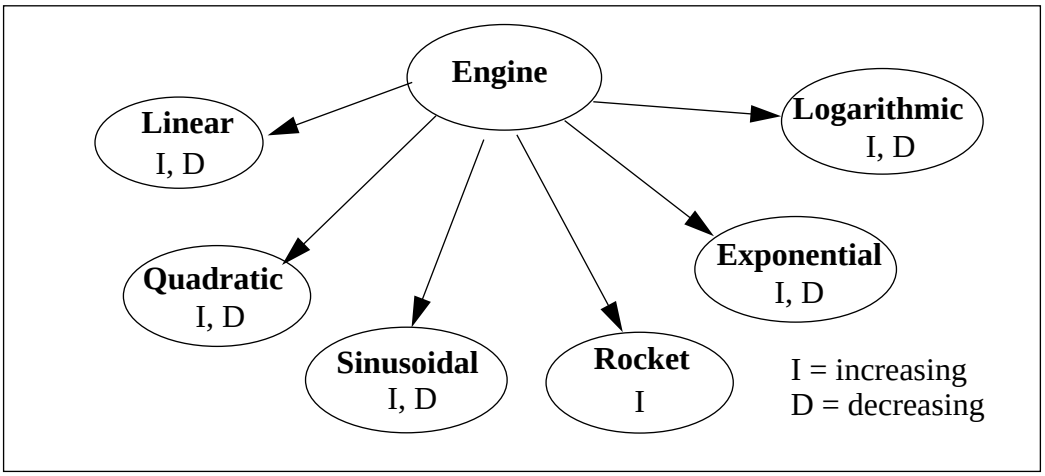

Figure 15: Engine Classes

C. IMPLEMENTATION

Following are the symbols we use to describe the mathematical model for engine speed:

s = speed, which extends from 0 to s_{max}

r = range, which extends from 0 to r_{max}

a, b, c, d are constants

Our goal is to find the simplest equations that exhibit the characteristics of increasing and decreasing speed for each subclass. Those equations must satisfy the two initial conditions, (0, 0) and s_{max},r_{max}.

We use r as an input number which, when processed through the appropriate equation, maps to a value in the speed range (Figure 16). For example, with equation $s = 2r$, if r equals 2, then s is 4. In the code, the toolkit user chooses engine type and specifies s_{max} and r_{max}. After that, the appropriate s is returned for any given r, and vice versa.

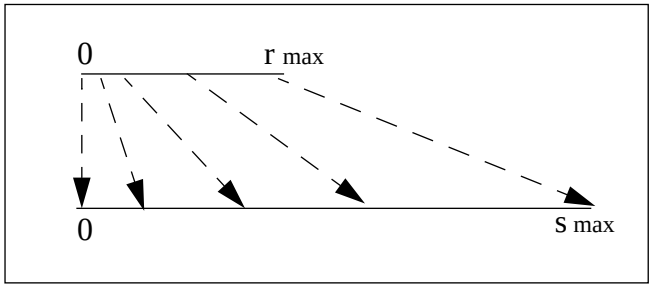

Figure 16: Range to speed mapping

Following is the derivation of possible equations to represent increasing and decreasing speed:

1. Increasing and decreasing speed

a. Linear

In this case, engine speed follows a simple linear equation. Consider the linear equation for speed

$$s - a = slope\,(r - b) \qquad \text{Eq 55}$$

Where slope is the slope of the linear curve. In this case, with two initial conditions, (0, 0) and (s_{max},r_{max}), our speed equation is (See Figure 17 and Eq 49):

$$s = \frac{s_{max}}{r_{max}} r \qquad \text{Eq 56}$$

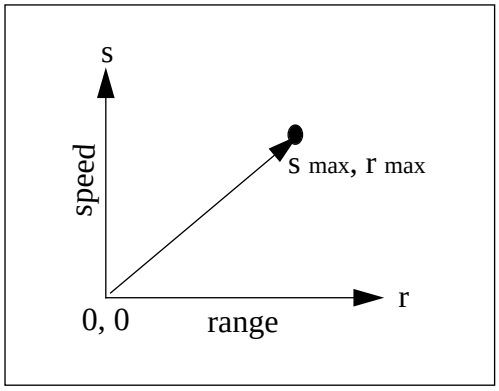

Figure 17: Linear approximation

The range equation is:

$$r = \frac{r_{max}}{s_{max}} s \qquad \text{Eq 57}$$

b. Quadratic

In this case, the speed equation can follow two quadratic curves--the solid and dotted lines in Figure 18--to increase or decrease speed.

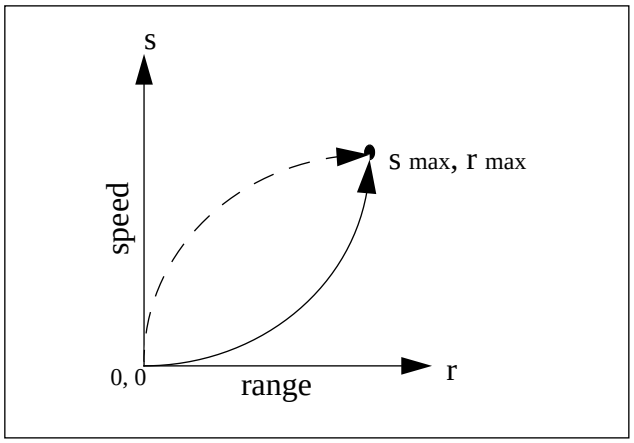

Figure 18: Quadratic approximation

We can derive the speed equation for the solid line as follows:

With the general quadratic equation $s = ar^2$, then according to Figure 10 with our two initial conditions (0, 0) and (s_{max}, r_{max}), we have $s_{max} = a(r_{max})^2$ so $a = \frac{s_{max}}{(r_{max})^2}$. Then the speed equation is:

$$s = \frac{s_{max}}{(r_{max})^2} r^2 \qquad \text{Eq 58}$$

The equation for the range is:

$$r = \sqrt{\frac{(r_{max})^2}{s_{max}} s} \qquad \text{Eq 59}$$

We can also derive the speed equation for the dotted line as follows.The general quadratic equation is $s - s_{max} = -a(r - r_{max})^2$. With two initial conditions, (0, 0) and (s_{max}, r_{max}), then $a = \frac{s_{max}}{(r_{max})^2}$. The speed equation is:

$$s - s_{max} = -\frac{s_{max}}{(r_{max})^2} (r - r_{max})^2 \qquad \text{Eq 60}$$

The range equation is:

$$r = -\sqrt{\frac{(r_{max})^2}{s_{max}} (s_{max} - s)} + r_{max} \qquad \text{Eq 61}$$

Speed can increase in a higher degree polynomial but, though straightforward, is not implemented here owing to lack of necessity. The general format for coding and using the code would be the same.

c. Logarithmic

In this case, speed increases along a logarithmic curve. The general equation for this case is:

$$s = a\log(1 + r) \qquad \text{Eq 62}$$

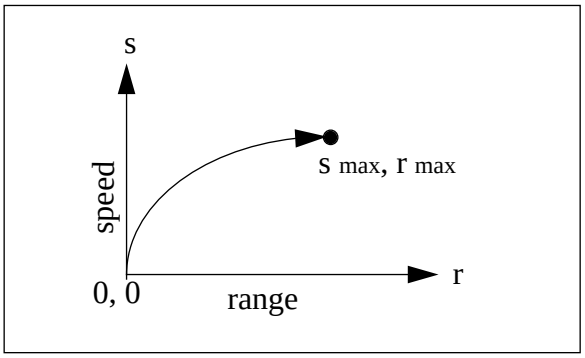

Figure 19: Logarithmic approximation

Based on Eq19 and two initial conditions, (0, 0) and (s_{max}, r_{max}), we derive $0 = a\log 1$, which is true and $s_{max} = a\log(1 + r_{max})$ then $a = \dfrac{s_{max}}{\log(1 + r_{max})}$. So our speed equation is:

$$s = \frac{s_{max}}{\log(1 + r_{max})}\log(1 + r) \qquad \text{Eq 63}$$

The range equation is:

$$r = e^{\left(\frac{s}{s_{max}}\log(1 + r_{max})\right)} - 1 \qquad \text{Eq 64}$$

d. Exponential

In this case, speed increases along an exponential curve. The general equation is $s + b = e^{ar}$, with two initial conditions of (0, 0) and (s_{max}, r_{max}). We derive $b = 1$ and $s_{max} + 1 = e^{ar_{max}}$ then $a = \frac{\log(s_{max} + 1)}{r_{max}}$. So our equation for the speed is:

$$s + 1 = e^{\frac{\log(s_{max} + 1)}{r_{max}} r} \qquad \text{Eq 65}$$

The range equation is:

$$r = \frac{r_{max}}{\log(s_{max} + 1)}\log(s + 1) \qquad \text{Eq 66}$$

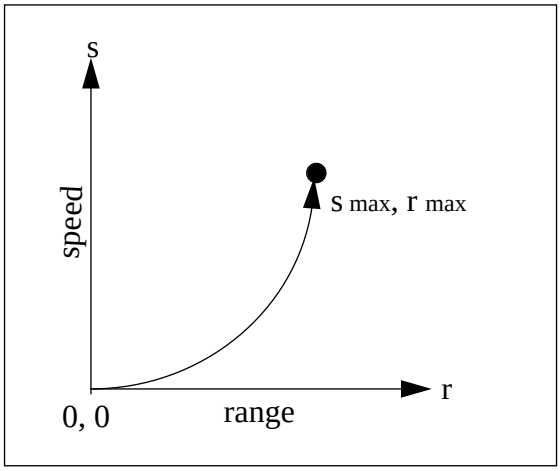

Figure 20: Exponential approximation

e. Sinusoidal

In this case, speed follows the sinusoidal curve. See Figure 13.

$$s = s_{max} \sin\theta \qquad \text{Eq 67}$$

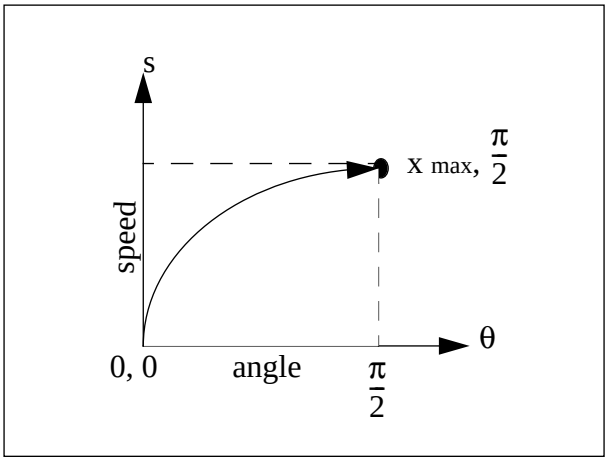

Figure 21: Sinusoidal approximation

Where $\theta \in \left[0, \frac{\pi}{2}\right]$, we can map input $(0, r_{max})$ to θ easily with equation $\theta = \frac{r}{r_{max}}\left(\frac{\pi}{2}\right)$. The range equation is:

$$r = \frac{2}{\pi} r_{max} \operatorname{asin} \frac{s}{s_{max}} \qquad \text{Eq 68}$$

f. Rocket

In this case, the rocket model in Chapter 4 governs the behavior of the speed curve. In general, the speed equation for the rocket is:

$$s = -u \ln\left(1 - \frac{rt}{m_o}\right) \qquad \text{Eq 69}$$

Where s = speed, u = fuel speed, t = rate of burning fuel, r = range of input, m_o = total mass of the object, and rt = mass of the fuel.

Figure 22 describes the behavior of the speed curve. The speed increases very rapidly to a very large value. Initial conditions are $(0,0)$ and (s_{max},r_{max}).

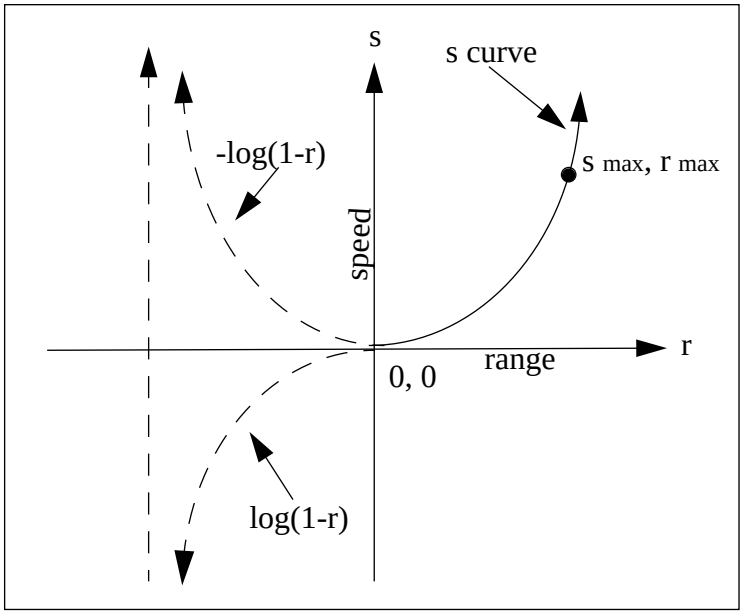

Figure 22: Rocket approximation

We use Eq 62 to implement the code as follows. The programmer supplies as input u, m_o, r, and r_{max}. From these initial values and by controlling the u value, we assign s_{max} to a value we want. With known values of s_{max}, r_{max} and the two equations, Eq 62 and Eq 63, we can now simulate the rocket engine.

The range equation is:

$$r = \frac{m_o}{t}(1 - e^{-s/u}) \qquad \text{Eq 70}$$

2. Increasing and decreasing speed on different curves

Usually, for the sake of simplicity, the decreasing speed curve for an engine is the same as the increasing case. But we can always use different speed curves on the same engine. One of the problems we encountered is in how to represent the input range of the throttle to the user. If we use the two curves in Figure 23, then for one value of s1 we have two range values, r1 and r2. Which range value, then, do we return to the user? One solution is to only return the r value of the increasing speed curve. Then when engine speed decreases, we map the decreasing speed to the equivalent increasing speed, and from that speed value and the range equation of the increasing case, we can find value r.

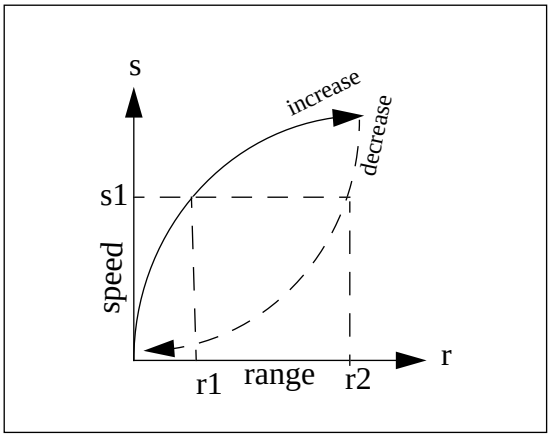

Figure 23: Decrease on different path

D. RESULTS

The code is organized into a base class and dependent subclasses. The base class holds the variables the other subclasses will use, such as maximum speed, maximum range, speed, and range. Each subclass has two central functions. One, *get_speed()*, returns the speed for a given range value, and the other, *get_range()*, returns the range for a given speed.

To incorporate the code to use for each case, the user includes a specific set of files. For example, to use the linear engine curve, include:

- engine.cpp
- linear.cpp (the name of the curve to use)
- a main function

The code was tested separately for each curve. Two loops were created for return values of speed and range. For return speed values, the range value extended from 0 to 100 in steps of 10. For range return values, the speed value extended from 0 to 1000 in steps of 100. See the code in Appendix C for details.

For the rocket engine model, the programmer should be careful with the given value of fuel speed u, rate of burning fuel t, and the total mass m_o. Remember that tr_{max} is the total fuel, it should be about half of the total mass m_o. The best value for maximum range r_{max} is about 100, and use a fuel speed value u to control the maximum speed value s_{max}.

The return values for each case behave according to the name of the case. For example, the linear model returns values in both speed and range linearly, and the exponential model returns speed values exponentially and its range values logarithmically as its equations describe.

In general, all models worked very well. The particular case is the rocket engine. With its physically-based model, it should be used for all jet engines in the simulation to enhance the realistic effects of our simulated world.

With the division of separate subclasses branching from the base class, the usage, testing, and enhancement of the code is straightforward. In summation, the models work very well in our simulated world.

VI. RESULTS AND CONCLUSIONS

A. RESULTS

The divide-and-conquer method was used to code and test the implementation described in this thesis, that is, each class was written in a separate file, segregated into a self-contained chunk of code. Therefore, potential problems and bugs were localized and the development process was reduced to small, manageable, straightforward steps. Once a particular class was validated, it required no further examination. Its derived classes did not return further problems in exchange for the benefits they received. Instead, they gained the benefit of being built onto a solid foundation. In general, the code is reliable, performs well, and is easy to use. All of the interface functions have a standardized format, and follow a standard naming convention.

1. Dead Reckoning

The design and implementation of the nine DR algorithms in this thesis are complete in the sense that they cover all possible cases in both world and body coordinates. In each perspective coordinate system, the implemented algorithms cover cases ranging from the simplest, a fixed entity, to the most detailed, an entity with rotation, velocity, and acceleration.

Most equations used to compute the results are simple, except for body coordinate algorithms 7 and 8, which are somewhat long and complex. However, all algorithms were coded with an efficiency as near to optimal as possible. For more examination of the applicable uses of these algorithms, consult the ARPA study for more details. [IST93]

2. Munitions

The munitions class in this thesis formed a basis for most of the munitions. It accounts for the bullet, bomb, ballistic, artillery, rocket, and guided missile. Further, the user can combine these munitions to form different types of weapon with each distinct munition performing a single phase of some composite weapon. For example, the bomb

model could be used to drop a missile from an airplane, and then when it reaches a certain altitude, the guided missile model could take over to govern a final stage wherein the missile seeks a target.

The guided missile mode of operation can be tail-chase or predicted-intercept depending on what kind of target position the user provides to compute the missile velocity vector. If the user provides the current position of the target, then the missile mode will be tail-chase, because the target will have moved to the next position at the next delta time. If the user provides the next dead reckoned position of the target, the missile mode will be the predicted-intercept missile.

Furthermore, the guided missile class can couple with any of the engine models in Chapter 5. Recall that the direction of the missile is based on the position of the missile and the position of the target. If the speed from the engine equation for every delta time is supplied to the munition, then the missile will chase after the target with the increasing speed of the engine curve.

The rocket model is based on Newton's equation. Its velocity and position equation are final. Together with the bomb equations, the rocket equations will form the basis for the ballistic missile model. However, the ballistic missile model was not implemented due to time constraints and its complexity.

3. Engine

Except for the rocket model which is physically-based, the remaining engine models are mathematically-based. But thorough testing of all the models reveals negligible difference between physically- and mathematically-based models.

In physically-based models, we use Newton's force law to get a mathematical expression of the speed curve, whereas in mathematical models, speed curves are encoded based on popular mathematical equations. In both cases, the initial condition (0, 0), the maximum speed, and the maximum range are used as boundary values. Our model becomes a boundary value problem, that is, the fitting of a curve between two points.

In testing, since the quadratic or exponential curve behaves very much the same as the logarithmic rocket curve, we can use them to substitute for the rocket speed curve. Overall, our models behaved realistically when tested.

B. CONCLUSIONS

The code described in this thesis keeps each separate class independent from each other. It is short and the performance fast. The names of the functions are standardized, making the toolkit easy to use. The toolkit was tested thoroughly to ensure reliability.

The work of this thesis cannot be considered groundbreaking, though. But it does craft existing knowledge into an organized and coherent piece of work. Its design and implementation are well thought out, and its performance in the current system is excellent. Furthermore, it lays a solid foundation for further study.

C. SUGGESTIONS FOR FUTURE WORK

After building the toolkit, the advantage of concise, independent functions that can be coupled with any entity models, was clear. We suggest that the work for suspension, control, steering models, and other dynamic entity attributes, be implemented in a manner consistent with the toolkit.

LIST OF REFERENCES

[BOYC77] Boyce, W., *Elementary Differential Equations*, 3rd Ed., John Wiley & Sons, Inc., Canada, 1977.

[HALL78] Halliday, D., *Physics*, 3rd Ed., John Wiley & Sons, Inc., Canada, 1978.

[IEEE93] Institute of Electrical and Electronics Engineers, International Standard, ANSI/IEEE Std. 1278-1993, *Standard for Information Technology, Protocols for Distributed Interactive Simulation*, March, 1993.

[IST93] Institute for Simulation and Training, *Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications*, Version 2.0 Fourth Draft, IST-CR-93-40, Orlando, Florida, Feb, 1994.

[MART84] Martin, W., *Elementary Differential Equations*, pp. 26-30, Dover Publications, Inc., New York, 1984.

[NATO] NATO, NATO Standardization Agreement (STANAG 4355), The Modified Point Mass Trajectory Model, NATO Headquarters, Brussels, Belgium.

[PARK92] Park, H., *NPSNET: Real-Time 3D Ground_Based Vehicle Dynamics*, Master Thesis, Naval Postgraduate School, Mar 1992.

[PRAT93] Pratt, D., *A software Architecture for the Construction and Management of Real-time Virtual Worlds*, PHD Dissertation, Computer Science Department, Naval Postgraduate School, Jun, 1993.

[PRAT94] Pratt, D., Zyda, M., Kelleher, K., *1994 Annual Report for the NPSNET Research Group*, Computer Science Department, Naval Postgraduate School, 1994.

[TOWE94] Towers, J., *Highly Dynamic Vehicles in a Real/Simulated Virtual Environment, Equations of Motion of The DIS 2.0.3 Dead Reckonings Algorithms*, ARPA, 1994.

[TRIM83] Trim, D., *Calculus and Analytic Geometry*, Addison-Wesley Publishing Company, Inc., Canada, 1983.

[ZESW93] Zeswitz, S., *NPSNET Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Interchange*, Master Thesis, Computer Science Department, Naval Postgraduate School, Sep, 1993.

INITIAL DISTRIBUTION LIST

1. Dudley Knox Library, Code 052 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

2. Chairman, Code 32 CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

3. Dr. Michael J. Zyda, Code CS/Zk 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. David R. Pratt, Code CS/Pr 10
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. John S. Falby, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Paul T. Barham, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000